

ADANA SCIENCE AND TECHNOLOGY UNIVERSITY

Introduction to Computer Programming II

Objectives for today

- Pointers and arrays
- Pointer arithmetics

POINTERS AND ARRAYS

Pointers and arrays

- The concept of arrays is related to that of pointers.
- In fact, arrays work very much like pointers to their first elements, and, actually, ***an array can always be implicitly converted to the pointer of the proper type***.
- For example, consider these two declarations:
  - int myarray [20];
  - int * mypointer;
- The following assignment operation would be valid:
  - mypointer = myarray;
  - Because they are both of type int *.

Pointers and arrays

- Pointers and arrays support the same set of operations, with the same meaning for both.
- The main difference being that
  - pointers can be assigned new addresses, while arrays cannot.
- For arrays, brackets ([]) were explained as specifying the index of an element of the array.
- Well, in fact these brackets are a dereferencing operator known as *offset operator*.
- They dereference the variable they follow just as *does, but they also add the number between brackets to the address being dereferenced.

Pointers and arrays

- For example:
  - a[5] = 0; // a [offset of 5] = 0
  - *(a+5) = 0; // pointed to by (a+5) = 0
- These two expressions are equivalent and valid, not only if a is a pointer, but also if a is an array.
- Remember that if an array, its name can be used just like **a pointer to its first element**.
- A reminder:
  - pointers can be assigned new addresses, while arrays cannot.
  - Meaning: Arrays are *constant* pointers.

Example

- Here is an example that mixes arrays and pointers:

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

```
10, 20, 30, 40, 50,
```

Pointer Initialization

- Pointers can be initialized to point to specific locations at the very moment they are defined:
  - int myvar;
  - int * myptr = &myvar;
- The resulting state of variables after this code is the same as after:
  - int myvar;
  - int * myptr;
  - myptr = &myvar;
- When pointers are initialized, what is initialized is the address they point to (i.e., myptr), never the value being pointed (i.e., *myptr).

Pointer initialization

- Therefore, the code above shall not be confused with:
  - int myvar;
  - int * myptr;
  - *myptr = &myvar;
- Which anyway would not make much sense **(and is not valid code).**
  - **myptr* is of type **int** whereas *&myvar* is of type **int ***.
    - They do not have the same type and automatic conversion is not possible.

Pointer arithmetics

- To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer types.
- To begin with, only addition and subtraction operations are allowed; the others make no sense in the world of pointers.
- But both addition and subtraction have a slightly different behavior with pointers, according to the size of the data type to which they point.

Pointer arithmetics

- When fundamental data types were introduced, we saw that types have different sizes.
- For example: char always has a size of 1 byte, int has 4, etc.

```
int main()
{
    int a;
    long b;
    char c;
    double d;
    float f;
    cout<<"Size of variables with different types : "<<endl;
    cout<<"-------------------------------------------"<<endl;
    cout<<"Size of type int \t: "<<sizeof(a)<<endl;
    cout<<"Size of type long \t: "<<sizeof(b)<<endl;
    cout<<"Size of type char \t: "<<sizeof(c)<<endl;
    cout<<"Size of type double \t: "<<sizeof(d)<<endl;
    cout<<"Size of type float \t: "<<sizeof(f)<<endl;
    cout<<"-------------------------------------------"<<endl;
    return 0;
}
```

```
Size of varaibles with different types :
------------------------------------------
Size of type int        : 4
Size of type long       : 4
Size of type char       : 1
Size of type double     : 8
Size of type float      : 4
------------------------------------------
```

```
int main()
{
    int *aPtr;
    long *bPtr;
    char *cPtr;
    double *dPtr;
    float *fPtr;
    cout<<"Size of pointers with different types : "<<endl;
    cout<<"------------------------------------------"<<endl;
    cout<<"Size of type int* \t: "<<sizeof(aPtr)<<endl;
    cout<<"Size of type long* \t: "<<sizeof(bPtr)<<endl;
    cout<<"Size of type char* \t: "<<sizeof(cPtr)<<endl;
    cout<<"Size of type double* \t: "<<sizeof(dPtr)<<endl;
    cout<<"Size of type float* \t: "<<sizeof(fPtr)<<endl;
    cout<<"------------------------------------------"<<endl;
    return 0;
}
```

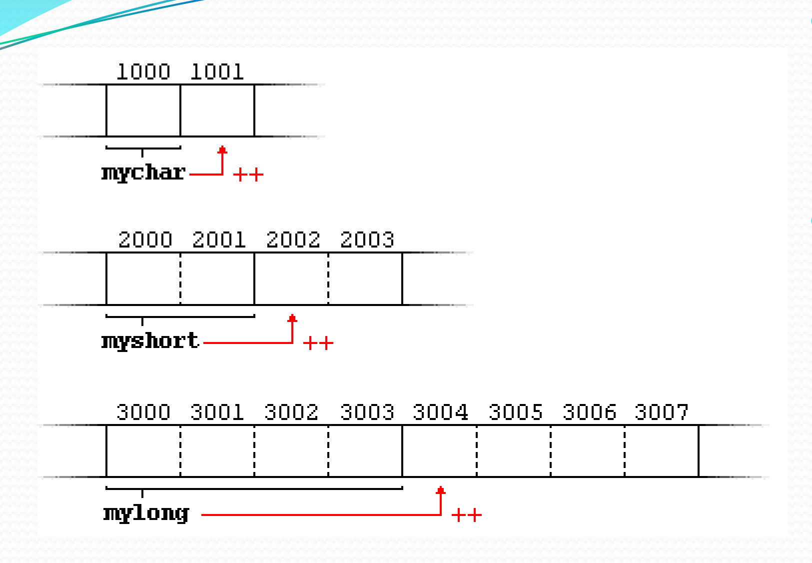
```
Size of pointers with different types :
------------------------------------------
Size of type int*        : 4
Size of type long*       : 4
Size of type char*       : 4
Size of type double*     : 4
Size of type float*      : 4
------------------------------------------
```

Pointer arithmetics

- Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.
- pointer arithmetic—certain arithmetic operations may be performed on pointers:
  - increment (++)
  - decremented (- -)
  - an integer may be added to a pointer (+ or +=)
  - an integer may be subtracted from a pointer (- or -=)
  - one pointer may be subtracted from another of the same type

Pointer arithmetics

- Suppose now that we define three pointers in this compiler:
  - char *mychar;
  - short *myshort;
  - long *mylong
- and that we know that they point to the memory locations 1000, 2000, and 3000, respectively.
- Therefore, if we write:
  - ++mychar;
  - ++myshort;
  - ++mylong;
- mychar, as one would expect, would contain the value 1001.
- But not so obviously, myshort would contain the value 2002.
- and mylong would contain 3004.
  - even though they have each been incremented only once.
- The reason is that, when adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer.



- This is applicable both when adding and subtracting any number to a pointer.
- It would happen exactly the same if we wrote:
  - mychar = mychar + 1;
  - myshort = myshort + 1;
  - mylong = mylong + 1;

- Regarding the increment (++) and decrement (--) operators, they both can be used as either prefix or suffix of an expression, with a slight difference in behavior:
  - as a prefix, the increment happens before the expression is evaluated,
  - as a suffix, the increment happens after the expression is evaluated.

Pointer arithmetics

- This also applies to expressions incrementing and decrementing pointers, which can become part of more complicated expressions that also include dereference operators (*).
- Remembering operator precedence rules, we can recall that <u>postfix operators, such as increment and decrement, have higher precedence than prefix operators</u>, such as the dereference operator (*).
- Therefore, the following expression:
  - *p++
- is equivalent to *(p++).

Pointer arithmetics

- *(p++) (or simply *p++) increases the value of p (so it now points to the next element),
  - but because ++ is used as postfix, the whole expression is evaluated as the value pointed originally by the pointer (the address it pointed to before being incremented).
    - First use then increase

```
int a[]= {1,2,3,4};
int *ptr = a;
for (int i=0; i<sizeof(a)/sizeof(a[0]); i++)
    cout<<"&a["<<i<<"] : "<<&a[i]<<endl;
```

Name	a[0]	a[1]	a[2]	a[3]	ptr
Address	0x28fefc	0x28ff00	0x28ff04	0x28ff08	0x28fef8
Value	1	2	3	4	0x28fefc

```
cout<<"&ptr : "<<&ptr<<endl;
cout<<"ptr : "<<ptr<<endl;
cout<<"*ptr : "<<*ptr<<endl;
cout<<"*ptr++ : "<<*ptr++<<endl;
cout<<"ptr : "<<ptr<<endl;
```

```
&a[0] : 0x28fefc
&a[1] : 0x28ff00
&a[2] : 0x28ff04
&a[3] : 0x28ff08
&ptr : 0x28fef8
ptr : 0x28fefc
*ptr : 1
*ptr++ : 1
ptr : 0x28ff00
```

Since ++ is used as postfix first the value 0x28fefc is used for printing the value (*p) then the value of ptr is changed to 0x28ff00.

Pointer arithmetics

- Essentially, these are the four possible combinations of the dereference operator with both the prefix and suffix versions of the increment operator (the same being applicable also to the decrement operator):
  - `*p++`
    - `// same as *(p++): increment pointer, and dereference` **`un`**`incremented address`
  - `*++p`
    - `// same as *(++p): increment pointer, and dereference incremented address`
  - `++*p`
    - `// same as ++(*p): dereference pointer, and increment the value it points to`
  - `(*p)++`
    - `// dereference pointer, and post-increment the value it points to`

Pointer Expressions and Pointer Arithmetic (cont.)

- Pointers can be compared using equality and relational operators.
  - Comparisons using relational operators are meaningless unless the pointers point to members of the same array.
  - Pointer comparisons compare the addresses stored in the pointers.
- A common use of pointer comparison is determining whether a pointer is 0 (i.e., the pointer is a null pointer—it does not point to anything).

Pointers

and const

Pointers and const

- Pointers can be used to access a variable by its address, and this access may include modifying the value pointed.
- But it is also possible to declare pointers that can access the pointed value to read it, but not to modify it.
- For this, it is enough with qualifying the type pointed to by the pointer as const.
- For example:

```
int x;
int y = 10;
const int * p = &y;
x = *p;          // ok: reading p
*p = x;          // error: modifying p, which is const-qualified
```

Pointers and const

```
int x;
int y = 10;
const int * p = &y;
x = *p;          // ok: reading p
*p = x;          // error: modifying p, which is const-qualified
```

- Here p points to a variable, but points to it in a const-qualified manner,
  - meaning that it can read the value pointed, but it cannot modify it.
- Note also, that the expression &y is of type int*, but this is assigned to a pointer of type const int*.
- This is allowed:
  - a pointer to non-const can be implicitly converted to a pointer to const.
  - But not the other way around!
    - As a safety feature, pointers to const are not implicitly convertible to pointers to non-const.

Pointers and const

- One of the use cases of pointers to const elements is as function parameters:
  - a function that takes a pointer to non-const as parameter can modify the value passed as argument,
  - while a function that takes a pointer to const as parameter cannot.
- Note that print_all uses pointers that point to constant elements.
- These pointers point to constant content they cannot modify,
  - but they are not constant themselves: i.e., the pointers can still be incremented or assigned different addresses, although they cannot modify the content they point to.

```cpp
// pointers as arguments:
#include <iostream>
using namespace std;

void increment_all (int* start, int* stop)
{
  int * current = start;
  while (current != stop) {
    ++(*current);  // increment value pointed
    ++current;     // increment pointer
  }
}

void print_all (const int* start, const int* stop)
{
  const int * current = start;
  while (current != stop) {
    cout << *current << '\n';
    ++current;     // increment pointer
  }
}

int main ()
{
  int numbers[] = {10,20,30};
  increment_all (numbers,numbers+3);
  print_all (numbers,numbers+3);
  return 0;
}
```

```
11
21
31
```

Using const with Pointers (cont.)

- There are four ways to pass a pointer to a function
  - a nonconstant pointer to nonconstant data
  - a nonconstant pointer to constant data (Fig. 7.10)
  - a constant pointer to nonconstant data (Fig. 7.11)
  - a constant pointer to constant data (Fig. 7.12)
- Each combination provides a different level of access privilege.

Using const with Pointers (cont.)

- The highest access is granted by a nonconstant pointer to nonconstant data
  - The data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data.
- Such a pointer's declaration (e.g., `int *countPtr`) does not include `const`.

Using const with Pointers (cont.)

- A nonconstant pointer to constant data
  - A pointer that can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified through that pointer.
- Might be used to receive an array argument to a function that will process each array element, but should not be allowed to modify the data.
- Any attempt to modify the data in the function results in a compilation error.
- Sample declaration:
  - `const int *countPtr;`
  - Read from right to left as "`countPtr` is a pointer to an integer constant."
- Figure 7.10 demonstrates the compilation error messages produced when attempting to compile a function that receives a nonconstant pointer to constant data, then tries to use that pointer to modify the data.

```cpp
1  // Fig. 7.10: fig07_10.cpp
2  // Attempting to modify data through a
3  // nonconstant pointer to constant data.
4
5  void f( const int * ); // prototype
6
7  int main()
8  {
9     int y;
10
11    f( &y ); // f attempts illegal modification
12 } // end main
13
14 // xPtr cannot modify the value of constant variable to which it points
15 void f( const int *xPtr )
16 {
17    *xPtr = 100; // error: cannot modify a const object
18 } // end function f
```

Fig. 7.10 | Attempting to modify data through a nonconstant pointer to constant data.

GNU C++ compiler error message:

```
fig07_10.cpp: In function `void f(const int*)':
fig07_10.cpp:17: error: assignment of read-only location
```

Using const with Pointers (cont.)

- A constant pointer to nonconstant data is *a pointer that always points to the same memory location*; the data at that location can be modified through the pointer.
- An example of such a pointer is an array name, which is a constant pointer to the beginning of the array.
- All data in the array can be accessed and changed by using the array name and array subscripting.
- A constant pointer to nonconstant data can be used to receive an array as an argument to a function that accesses array elements using array subscript notation.
- Pointers that are declared `const` must be initialized when they're declared.
- If the pointer is a function parameter, it's initialized with a pointer that's passed to the function.



Common Programming Error 7.6

Not initializing a pointer that's declared `const` *is a compilation error.*

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int x = 5, y;
7
8  // ptr is a constant pointer to a constant integer.
9  // ptr always points to the same location; the integer
10 // at that location cannot be modified.
11     int *const ptr ;
12     ptr = &x;
13           cout << *ptr << endl;
14     *ptr = 7; // error: *ptr is const; cannot assign new value
15 //ptr = &y; // error: ptr is const; cannot assign new address
16 } // end main
```

```
      === Build: Debug in constPointers (compiler: GNU GCC Compiler) ===
      In function 'int main()':
11    error: uninitialized const 'ptr' [-fpermissive]
12    error: assignment of read-only variable 'ptr'
6     warning: unused variable 'y' [-Wunused-variable]
      === Build failed: 2 error(s), 1 warning(s) (0 minute(s), 0 second(s)) ===
```

```cpp
// Fig. 7.11: fig07_11.cpp
// Attempting to modify a constant pointer to nonconstant data.

int main()
{
   int x, y;

   // ptr is a constant pointer to an integer that can
   // be modified through ptr, but ptr always points to the
   // same memory location.
   int * const ptr = &x; // const pointer must be initialized

   *ptr = 7; // allowed: *ptr is not const
   ptr = &y; // error: ptr is const; cannot assign to it a new address
} // end main
```

Fig. 7.11 | Attempting to modify a constant pointer to nonconstant data.

GNU C++ compiler error message:

```
fig07_11.cpp: In function `int main()':
fig07_11.cpp:14: error: assignment of read-only variable `ptr'
```

Using const with Pointers (cont.)

- The minimum access privilege is granted by a constant pointer to constant data.
  - Such a pointer always points to the same memory location, and the data at that location cannot be modified via the pointer.
  - This is how an array should be passed to a function that *only reads the array, using array subscript notation, and does not modify the array.*
- The program of Fig. 7.12 declares pointer variable `ptr` to be of type `const int * const` (line 13).
- This declaration is read from right to left as "`ptr` is a constant pointer to an integer constant."
- The figure shows the error messages generated when an attempt is made to modify the data to which `ptr` points and when an attempt is made to modify the address stored in the pointer variable.

```
1  // Fig. 7.12: fig07_12.cpp
2  // Attempting to modify a constant pointer to constant data.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 5, y;
9
10    // ptr is a constant pointer to a constant integer.
11    // ptr always points to the same location; the integer
12    // at that location cannot be modified.
13    const int *const ptr = &x;
14
15    cout << *ptr << endl;
16
17    *ptr = 7; // error: *ptr is const; cannot assign new value
18    ptr = &y; // error: ptr is const; cannot assign new address
19 } // end main
```

Fig. 7.12 | Attempting to modify a constant pointer to constant data.

GNU C++ compiler error message:

```
fig07_12.cpp: In function `int main()':
fig07_12.cpp:17: error: assignment of read-only location
fig07_12.cpp:18: error: assignment of read-only variable `ptr'
```

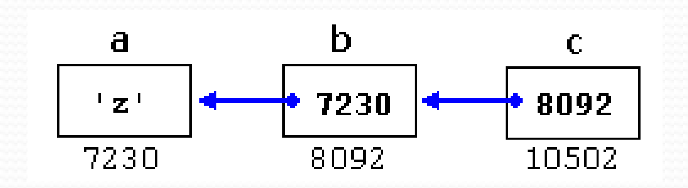
Pointers

to pointers

Pointers to pointers

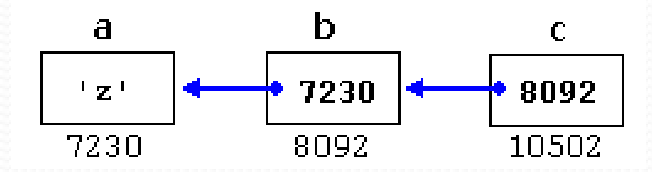
- C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers).
- The syntax simply requires an asterisk (*) for each level of indirection in the declaration of the pointer:

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```



- This, assuming the randomly chosen memory locations for each variable of 7230, 8092, and 10502, could be represented as:

Pointers to pointers



- With the value of each variable represented inside its corresponding cell, and their respective addresses in memory represented by the value under them.
- The new thing in this example is variable c, which is a pointer to a pointer, and can be used in three different levels of indirection, each one of them would correspond to a different value:
  - c is of type char** and a value of 8092
  - *c is of type char* and a value of 7230
  - **c is of type char and a value of 'z'

Pointers

void pointers

void Pointers

- The void type of pointer is a special type of pointer.
- In C++, void represents the *absence of type*.
- Therefore, void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).
- This gives void pointers a great flexibility, by being able to point to any data type, from an integer value or a float to a string of characters.
- In exchange, they have a great limitation: the data pointed to by them cannot be directly dereferenced (which is logical, since we have no type to dereference to),
  - and for that reason, any address in a void pointer needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.

void Pointers

- One of its possible uses may be to pass generic parameters to a function.
- For example:

```
// increaser
#include <iostream>
using namespace std;

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data; ++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << '\n';
  return 0;
}
```

```
y, 1603
```

void Pointers - important points

- All pointer types can be assigned to a pointer of type `void *` without casting.
- A `void *` pointer cannot be dereferenced.
  - The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer—for a pointer to `void`, this number of bytes cannot be determined from the type.

Invalid pointers and null pointers

- In principle, pointers are meant to point to valid addresses,
  - such as the address of a variable or the address of an element in an array.
- But pointers can actually point to any address, including addresses that do not refer to any valid element.
- Typical examples of this are *uninitialized pointers* and pointers to nonexistent elements of an array:

```
int * p;               // uninitialized pointer (local variable)

int myarray[10];
int * q = myarray+20;  // element out of bounds
```

Invalid pointers and null pointers

```
int * p;               // uninitialized pointer (local variable)

int myarray[10];
int * q = myarray+20;  // element out of bounds
```

- Neither p nor q point to addresses known to contain a value, but none of the above statements causes an error.
- In C++, pointers are allowed to take any address value, no matter whether there actually is something at that address or not.
- What can cause an error is to dereference such a pointer (i.e., actually accessing the value they point to).
- Accessing such a pointer causes undefined behavior, ranging from an error during runtime to accessing some random value.

Invalid pointers and null pointers

- But, sometimes, a pointer really needs to explicitly point to nowhere, and not just an invalid address.
- For such cases, there exists a special value that any pointer type can take: the *null pointer value*. T
- his value can be expressed in C++ in two ways: either with an integer value of zero, or with the **NULL** keyword:
  - `int * p = 0;`
  - `int * q = NULL;`
- Here, both p and q are *null pointers*,
  - meaning that they explicitly point to nowhere,
  - and they both actually compare equal: all *null pointers* compare equal to other *null pointers*.

Invalid pointers and null pointers

- NULL is defined in several headers of the standard library, and is defined as an alias of some *null pointer* constant value (such as 0).
- Do not confuse *null pointers* with void pointers!
- A *null pointer* is a value that any pointer can take to represent that it is pointing to "nowhere",
- while a void pointer is a type of pointer that can point to somewhere without a specific type.
- One refers to the value stored in the pointer, and the other to the type of data it points to.


Q&A